

MAGAZINE

BSD

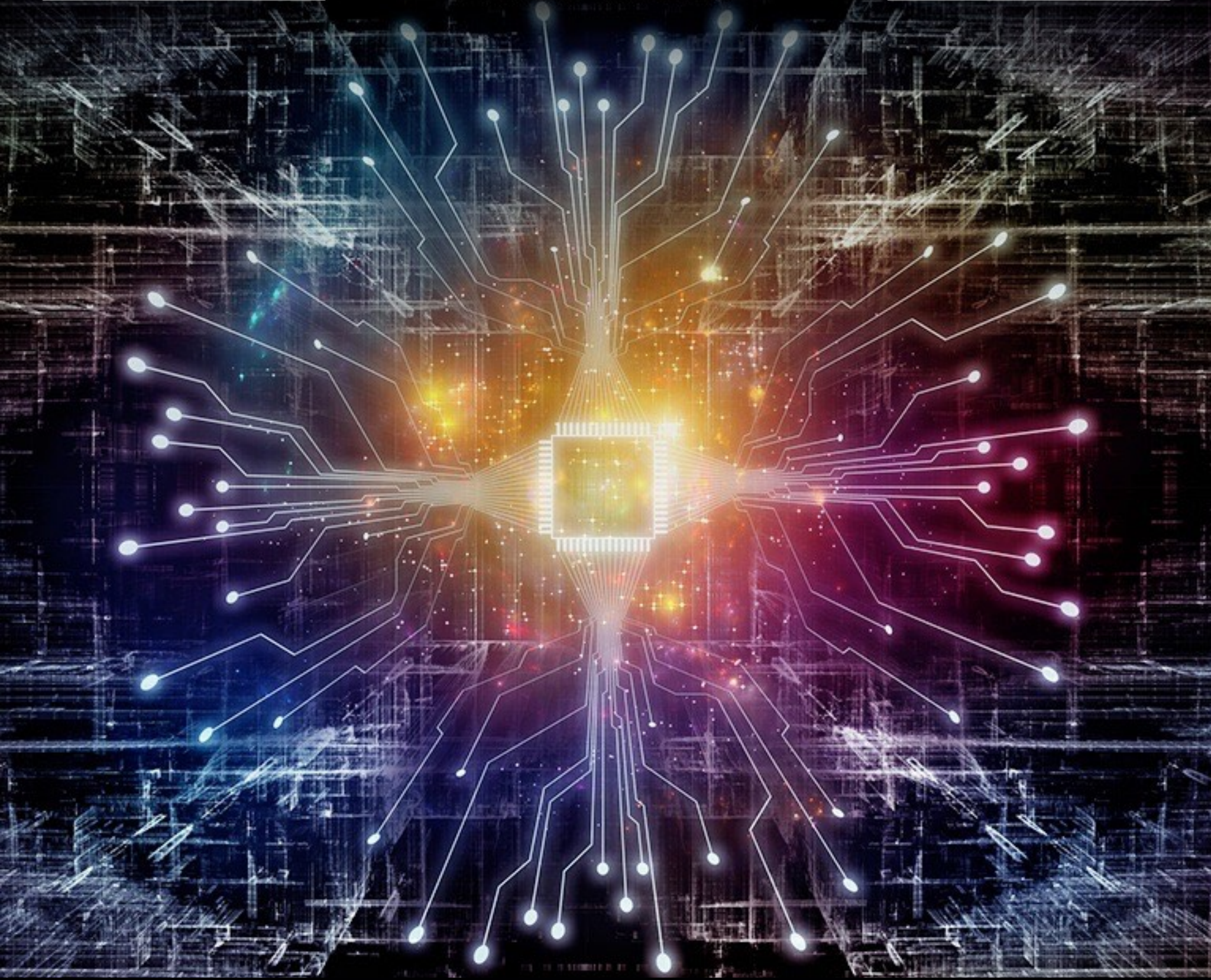
FOR NOVICE AND ADVANCED USERS

HOW TO BUILD A FREEBSD KERNEL MODULE FROM SCRATCH

**WORKSHOP
BY
DAVID CARLIER**

HOW TO BUILD A FREEBSD KERNEL MODULE FROM SCRATCH

DAVID CARLIER



WWW.BSDMAG.ORG

How to Build a FreeBSD Kernel Module From Scratch

This workshop was designed to help you understand how the userland communicates with the kernel through an existing example, studying the workflow; hence in the end you would be able to extend it or writing one of your own.

Module 1: FreeBSD Kernel Module

In this module, we will give an overview of the nature of the FreeBSD's kernel. The important configuration files will be explained in addition to learning how to compile the whole system with more options and with more debugging information enabled. This is very useful for kernel development.

Module 2: IPFW2 Userland and Kernel Workflow

In this module, we'll have an overview of ipfw2 - both userland and kernel side - and how they both interact.

Module 3: Through The Userland to Kernel Codes

In this module, we'll have an overview of ipfw2 - both userland and kernel side -, and how they interact. First of all, we will see how to use sysctl we saw in previous modules to set simple values. How to communicate settings to the kernel via a socket; all of it going through the userland to kernel codes.

Module 4: DUMMYNET Module Workflow Study

In this last module, we'll not only look at ipfw's communication with the kernel but also how the firewall configuration and rules are handled. We will go through the dummynet module, its workflow and how it operates with the kernel so you would be able to add new opcodes on your own.

David Carlier

*He is an experienced developer and used to handle some languages like C/C++, Java, Python with Linux, *BSD and Win32 Operating Systems and worked inside startups and bigger companies, too.*

Personally a big fan of FreeBSD and OpenBSD. C/C++ are his preferred programming language most of the time.

*He writes and reviews articles for BSDMag
<http://www.bsdmag.org>.*

He contributes modestly to OpenBSD ports and time in time to the source.

*He has been interviewed by BSDNow
show [http://www.bsdnow.tv/episodes/2017 10 18-software is storytelling](http://www.bsdnow.tv/episodes/2017_10_18-software_is_storytelling).*

He did some small contributions for FreeBSD and DragonflyBSD operating system.

COURSE CURRICULUM

Module 1: FreeBSD Kernel Module

06

Module 2: IPFW2 Userland and Kernel Workflow

18

Module 3: Through The Userland to Kernel Codes

33

Module 4: DUMMYNET Module Workflow Study

48

Module 1: FreeBSD

Kernel Module

In this module, we will give an overview of the nature of the FreeBSD's kernel. The important configuration files will be explained in addition to learning how to compile the whole system with more options and with more debugging information enabled. This is very useful for kernel development.

Requirements:

FreeBSD 10.x.

Machine with at least 4 cores is recommended for the system compilation.

Genuine hardware or virtualized environment as your convenience.

1/ The FreeBSD Kernel

FreeBSD, like many kernels, is a monolithic kernel with loadable module support. It is possible to build FreeBSD kernel with all needed modules statically, or for those modules that support it, as separated dynamic loadable modules.

Dynamic modules can be loaded and unloaded at will with `kldload/kldunload` or at boot time with `<name of kernel module>_load="YES"` in `/boot/loader.conf` file.

To have an overview of all currently loaded modules, you can type `kldstat`. For example, the output looks like the following:

```
Id Refs Address Size Name
1 19 0xffffffff80200000 19ff378 kernel
2 1 0xffffffff81e11000 4f62 ng_ubt.ko
...
9 1 0xffffffff81e5b000 3bab ng_socket.ko
```

Let's load the DTrace module by typing:

```
kldload dtraceall
```

Then if we type `kldstat` again, we should see some new entries related to this module:

...

```
10 1 0xffffffff81e5f000 89e dtraceall.ko
```

```
11 11 0xffffffff81e60000 9964 opensolaris.ko
```

```
12 10 0xffffffff81e6a000 857dba dtrace.ko
```

...

If we add `dtraceall_load="YES"` to `/etc/rc.conf`, we can use Dtrace framework facility after reboot. You can find an excellent introduction to Dtrace in the December 2016 issue of BSDmag.

(<http://bsdmag.org/download/samba-nfs-and-firewall-new-bsd-issue/>)

Indeed, Dtrace can be useful for tracing syscalls.

2/ Configuration

To build the system, we need the source code for both the kernel and userland. The userland is simply all the base utilities of FreeBSD. The kernel and userland code are consistently tied together, and available in the same subversion repository. Apart from pure BSD code, we can find GNU libraries and software (called contrib code). In addition, for ZFS, CTF (Compact C Type Format debug section, similar to DWARF format but reduced in terms of size) and DTrace proper compilations, some CDDL codes are present. Happily, FreeBSD is provided with subversion in base, suffixed distinctly to avoid colliding with the port version.

checkout the source in `/usr/src` via `svn` as a privileged user

`sudo` (or as root) `svn co https://svn0.us-east.freebsd.org/base/stable/10 /usr/src` (or you can checkout the current branch with much newer code but with more instability, you can just replace `stable/10` by `head`)

To better understand how the kernel options work, we will have a look at the `/usr/src/sys/conf/options` file.

Indeed, this file serves to indicate in the kernel level which symbols/constants from a certain header file we want to include in the build process.

```

...

# $FreeBSD$

#

...

#

#

# Format of this file:

# Option name filename

#

# If filename is missing, the default is

# opt_<name-of-option-in-lower-case>.h

AAC_DEBUG opt_aac.h

AACRAID_DEBUG opt_aacraid.h

AHC_ALLOW_MEMIO opt_aic7xxx.h

AHC_TMODE_ENABLE opt_aic7xxx.h

AHC_DUMP_EEPROM opt_aic7xxx.h

AHC_DEBUG opt_aic7xxx.h

AHC_DEBUG_OPTS opt_aic7xxx.h

AHC_REG_PRETTY_PRINT opt_aic7xxx.h

AHD_DEBUG opt_aic79xx.h

AHD_DEBUG_OPTS opt_aic79xx.h

AHD_TMODE_ENABLE opt_aic79xx.h

AHD_REG_PRETTY_PRINT opt_aic79xx.h

ADW_ALLOW_MEMIO opt_adw.h

...

```

There are up to two fields on each line: the option's name and the file created with the relevant preprocessor defined. If the option is present in your kernel configuration file, let's say `AHD_DEBUG_OPTS`, it is possible to test if `AHD_DEBUG_OPTS` is defined and providing some contextual code for this option.

Let's imagine we wrote a new shiny kernel module. We could add our proper line in this file.

```
BSDMAG opt_bsdmag.h
```

Another important file is `/usr/src/sys/conf/files`. This file serves to indicate which kernel module needs to be included in the build process.

```
...  
  
cam/cam.c optional scbus  
cam/cam_compat.c optional scbus  
cam/cam_periph.c optional scbus  
cam/cam_queue.c optional scbus  
cam/cam_sim.c optional scbus  
cam/cam_xpt.c optional scbus  
cam/ata/ata_all.c optional scbus  
cam/ata/ata_xpt.c optional scbus  
cam/ata/ata_pmp.c optional scbus  
cam/scsi/scsi_xpt.c optional scbus  
cam/scsi/scsi_all.c optional scbus  
cam/scsi/scsi_cd.c optional cd  
cam/scsi/scsi_ch.c optional ch  
cam/ata/ata_da.c optional ada | da  
cam/ctl/ctl.c optional ctl  
cam/ctl/ctl_backend.c optional ctl  
cam/ctl/ctl_backend_block.c optional ctl
```

```
cam/ctl/ctl_backend_ramdisk.c optional ctl
cam/ctl/ctl_cmd_table.c optional ctl
cam/ctl/ctl_frontend.c optional ctl
cam/ctl/ctl_frontend_cam_sim.c optional ctl
cam/ctl/ctl_frontend_internal.c optional ctl
cam/ctl/ctl_frontend_iscsi.c optional ctl
cam/ctl/ctl_scsi_all.c optional ctl
cam/ctl/ctl_tpc.c optional ctl
cam/ctl/ctl_tpc_local.c optional ctl
cam/ctl/ctl_error.c optional ctl
cam/ctl/ctl_util.c optional ctl
cam/ctl/scsi_ctl.c optional ctl
cam/scsi/scsi_da.c optional da
cam/scsi/scsi_low.c optional ct | ncv | nsp | stg
cam/scsi/scsi_pass.c optional pass
cam/scsi/scsi_pt.c optional pt
cam/scsi/scsi_sa.c optional sa
...
```

Each line has the following: the relative path to sys and the type of module. If type is optional; the module will be compiled with the (lower case) option name written afterwards.

Again, with our new module, we can add our specific kernel module C file. For example, for file `workshop_module1.c`:

```
workshop_bsdماغmodule1.c optional bsdماغ
```

3/ Build

So now we can create a custom kernel config. Let's call it WORKSHOP.

```
cp /usr/src/sys/<arch>/conf/GENERIC /usr/src/sys/<arch>/conf/WORKSHOP
```

```
echo "KERNCONF=WORKSHOP" >> /etc/make.conf (it will pick up the new WORKSHOP  
configuration file, by default it is the GENERIC one)
```

Steps to build a system:

First, the userland needs to be compiled.

Go to `/usr/src`

If your machine has multiple cores, it is advised to use them for the system compilation.

This might take several hours depending on your current configuration.

Build the userland:

```
make -j<number of cores+1> buildworld
```

Build the kernel:

```
make -j<number of cores+1> buildkernel
```

Install the kernel:

```
make installkernel (install the kernel in /)
```

It is possible to do the following to build userland and the kernel in the same command:

```
make -j<number of cores+1> buildworld kernel
```

Restart in single user mode (via command line shutdown -r or via the boot menu)

go to `/usr/src`

Then run merge master to merge configuration and other files from the current system with the new ones that were built.

```
mergemaster -p
```

Then, install:

```
make installworld
```

then:

```
mergemaster -FUi
```

Mergemaster will try to merge various configurations files and asking you how you wish to proceed, merging as possible, replacing with a newer one or keeping the existing file. Warning, doing so the system will attempt to merge config files in various places, especially regarding potential ssh, user/groups related, a bit of caution not deleting additional settings/users/groups in the process, merge master will always ask you how do you plan to merge via your preferred editor.

Restart in normal mode.

You should have now a workable system with the latest fixes/patches for the 10.x branch. But as a developer, we might need more info from the system for debugging, studying the core dump after a system crash/kernel panic. It is advisable to enable kernel core dump writing (could be enabled when you installed FreeBSD or, afterwards, can be enabled via the `dumpdir rc.conf` variable) at the cost of disk space consumption (can be potentially important, deleting old ones is necessary.). They are, by default, located in `/var/crash`. To debug a kernel crash dump, the kernel compiled with debugging symbols, `kernel.debug`, is necessary. `gdb` can be used in the following way.

```
kgdb
```

```
GNU gdb 6.1.1 [FreeBSD]
```

```
Copyright 2004 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are  
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for details.
```

```
This GDB was configured as "amd64-marcel-freebsd"...
```

Reading symbols from /boot/kernel/ng_ubt.ko.symbols...done.
Loaded symbols for /boot/kernel/ng_ubt.ko.symbols
Reading symbols from /boot/kernel/netgraph.ko.symbols...done.
Loaded symbols for /boot/kernel/netgraph.ko.symbols
Reading symbols from /boot/kernel/ng_hci.ko.symbols...done.
Loaded symbols for /boot/kernel/ng_hci.ko.symbols
Reading symbols from /boot/kernel/ng_bluetooth.ko.symbols...done.
Loaded symbols for /boot/kernel/ng_bluetooth.ko.symbols
Reading symbols from /boot/kernel/ng_l2cap.ko.symbols...done.
Loaded symbols for /boot/kernel/ng_l2cap.ko.symbols
Reading symbols from /boot/kernel/ng_btsocket.ko.symbols...done.
Loaded symbols for /boot/kernel/ng_btsocket.ko.symbols
Reading symbols from /boot/kernel/ng_socket.ko.symbols...done.
Loaded symbols for /boot/kernel/ng_socket.ko.symbols
Reading symbols from /boot/kernel/dtraceall.ko.symbols...done.
Loaded symbols for /boot/kernel/dtraceall.ko.symbols
Reading symbols from /boot/kernel/opensolaris.ko.symbols...done.
Loaded symbols for /boot/kernel/opensolaris.ko.symbols
Reading symbols from /boot/kernel/dtrace.ko.symbols...done.
Loaded symbols for /boot/kernel/dtrace.ko.symbols
Reading symbols from /boot/kernel/dtmalloc.ko.symbols...done.
Loaded symbols for /boot/kernel/dtmalloc.ko.symbols
Reading symbols from /boot/kernel/dtnfscl.ko.symbols...done.
Loaded symbols for /boot/kernel/dtnfscl.ko.symbols
Reading symbols from /boot/kernel/fbt.ko.symbols...done.
Loaded symbols for /boot/kernel/fbt.ko.symbols
Reading symbols from /boot/kernel/fasttrap.ko.symbols...done.

```

Loaded symbols for /boot/kernel/fasttrap.ko.symbols
Reading symbols from /boot/kernel/lockstat.ko.symbols...done.
Loaded symbols for /boot/kernel/lockstat.ko.symbols
Reading symbols from /boot/kernel/sdt.ko.symbols...done.
Loaded symbols for /boot/kernel/sdt.ko.symbols
Reading symbols from /boot/kernel/systrace.ko.symbols...done.
Loaded symbols for /boot/kernel/systrace.ko.symbols
Reading symbols from /boot/kernel/systrace_freebsd32.ko.symbols...done.
Loaded symbols for /boot/kernel/systrace_freebsd32.ko.symbols
Reading symbols from /boot/kernel/profile.ko.symbols...done.
Loaded symbols for /boot/kernel/profile.ko.symbols

#0 sched_switch (td=0xffffffff8011b80a940, newtd=<value optimized out>,
flags=-2123250552) at /usr/src/sys/kern/sched_ule.c:1940

1940 cpuid = PCPU_GET(cpuid);

```

Like the userland gdb's counterpart, we can use backtrace (bt).

```

(kgdb) backtrace

#0 sched_switch (td=0xffffffff8011b80a940, newtd=<value optimized out>,
flags=-2123250552) at /usr/src/sys/kern/sched_ule.c:1940

#1 0xffffffff8095b139 in mi_switch (flags=Unhandled dwarf expression opcode
0x93

) at /usr/src/sys/kern/kern_synch.c:492

#2 0xffffffff8099b172 in sleepq_switch (wchan=<value optimized out>,
pri=<value optimized out>) at /usr/src/sys/kern/subr_sleepqueue.c:552

#3 0xffffffff8099afd3 in sleepq_wait (wchan=0xffffffff80115316200, pri=Unhandled
dwarf expression opcode 0x93

) at /usr/src/sys/kern/subr_sleepqueue.c:631

```

```
#4 0xfffffffff8095aa47 in _sleep (ident=0x0, lock=0xffffffff80115316230,
priority=0, wmesg=0xfffffffff80ff47f2 "-", sbt=0, pr=0, flags=<value optimized
out>) at /usr/src/sys/kern/kern_synch.c:254

#5 0xfffffffff8099f778 in taskqueue_thread_loop (arg=<value optimized out>) at
/usr/src/sys/kern/subr_taskqueue.c:118

#6 0xfffffffff8091e234 in fork_exit (callout=0xfffffffff8099f6b0
<taskqueue_thread_loop>, arg=0xffffffff800078ebe90, frame=0xfffffe0232e9fac0) at
/usr/src/sys/kern/kern_fork.c:996

#7 0xfffffffff80d4f4fe in fork_trampoline () at
/usr/src/sys/amd64/amd64/exception.S:610

#8 0x0000000000000000 in ?? ()
```

Also, list if we want to see

```
(kgdb) list *0xfffffffff8095aa47 (coming from the frame number 4 from above)

0xfffffffff8095aa47 is in _sleep (/usr/src/sys/kern/kern_synch.c:254).

249 else if (sbt != 0)

250 rval = sleepq_timedwait(ident, pri);

251 else if (catch)

252 rval = sleepq_wait_sig(ident, pri);

253 else {

254 sleepq_wait(ident, pri);

255 rval = 0;

256 }

257 #ifdef KTRACE

258 if (KTRPOINT(td, KTR_CSW))
```

Then, for example, going up in the stack frames calls and so on ...

```
(kgdb) up 2

#4 0xfffffffff8095aa57 in _sleep (ident=0x0, lock=0xffffffff800035c6a30,
```

```

priority=0, wmesg=0xffffffff80ff47f2 "-", sbt=0, pr=0,
flags=<value optimized out>) at /usr/src/sys/kern/kern_synch.c:254
254 sleepq_wait(ident, pri);

(kgdb) list
249 else if (sbt != 0)
250     rval = sleepq_timedwait(ident, pri);
251 else if (catch)
252     rval = sleepq_wait_sig(ident, pri);
253 else {
254     sleepq_wait(ident, pri);
255     rval = 0;
256 }
257 #ifdef KTRACE
258 if (KTRPOINT(td, KTR_CSW))

```

For more information about gdb, a good exists at:

<http://bsdmag.org/course/application-debugging-and-troubleshooting-2>

If you run the -CURRENT branch, the kernel can crash for various reasons, and the gdb-like tool is handy to get a basic understanding of the reasons for the crash.

Detecting potential deadlocks.

FreeBSD does not rely on the Giant Lock model anymore. Instead, it has fine-grained locking/unlocking process. Hence, the resulting programming can be tricky and it is easy to get lock contentions. As a kernel developer, you can always enable the WITNESS* kernel options for detecting contentions and locks circular references; but beware that the system becomes pretty slow without WITNESS_SKIPSPIN (skip spin locks basically) is not activated.

Having read this workshop module, you learned the basics of kernel custom configuration, compiling the whole system.

Exercise

- To enable those additional capabilities, deadlock detections, more debugging info, and additional checking, which options in conf/options are needed?
- Recompile the kernel with those options.
- Once the system is restarted, which differences are you noticing? Eventually, what are the downsides?
- Is it possible to improve the situation? How?

Module 2: IPFW2

Userland and Kernel Workflow

In this module, we'll have an overview of ipfw2 - both userland and kernel side - and how they both interact.

1/ IPFW command line settings via sysctl

We can find it under the FreeBSD's source code we got from svn in the first module.

```
<source code root path>/sbin/ipfw/
```

In the previous module, we saw the base of a kernel module. IPFW works differently. It enables/disables features via sysctl. Those who have done some FreeBSD programming might have used it, so syscalls like sysctl / sysctlbyname / sysctlnametomib are already familiar, and you can jump directly to the next chapter.

Otherwise, IPFW uses sysctlbyname and sysctl. Here are their function signatures:

```
int sysctlbyname(const char *name, void *oldp, size_t *oldlenp, const void *newp, size_t newlen);
```

```
int sysctl(const int *name, u_int namelen, void *oldp, size_t oldlenp, const void *newp, size_t newlen);
```

If you wish to get a value, the oldp and oldlenp arguments need to be used. To set a value, use the newp and newlen arguments.

For example, to get the number of CPUs available:

```
int nbcpu;

size_t nbcpulen = sizeof(nbcpu);

...

if (sysctlbyname("hw.ncpu", &nbcpu, &nbcpulen, NULL, 0) == 0)

printf("%d cpus\n", nbcpu);
```

Alternatively:

```
int mib[2];

...

mib[0] = CTL_HW;

mib[1] = HW_NCPU;

...

if (sysctl(mib, sizeof(mib), &nbcpu, &nbcpulen, NULL, 0) == 0)

...
```

Or the opposite, setting a value, like the number of maximum file descriptors:

```
int maxfiles = 4096;

size_t maxfileslen = sizeof(maxfiles);

...

if (sysctlbyname("kern.maxfiles", NULL, 0, &maxfiles, maxfileslen) == 0)...

...

mib[0] = CTL_KERN;

mib[1] = KERN_MAXFILES;

...

if (sysctl(mib, sizeof(mib), NULL, 0, &maxfiles, maxfileslen) == 0)
```

...

IPFW uses the sysctl* family function to turn on and off the firewall itself or to make ipfw more verbose.

Here is the code where the firewall is enabled/disabled:

```
/* EXPLANATION: av here are the parameter passed by command line */
} else if (_substrcmp(*av, "firewall") == 0) {

    sysctlbyname("net.inet.ip.fw.enable", NULL, 0,
                &which, sizeof(which));

    sysctlbyname("net.inet6.ip6.fw.enable", NULL, 0,
                &which, sizeof(which));
```

2/ IPFW command line settings via socket

In addition, ipfw uses a socket to, for example, add a rule via an identified optname.

Here is a sample of the code responsible for getting settings from the kernel:

...

```
static int
table_do_modify_record(int cmd, ipfw_obj_header *oh,
ipfw_obj_tentry *tent, int count, int atomic)
{
    ipfw_obj_ctlv *ctlv;
    ipfw_obj_tentry *tent_base;
    caddr_t pbuf;

    char xbuf[sizeof(*oh) + sizeof(ipfw_obj_ctlv) + sizeof(*tent)];
```

```

int error, i;

size_t sz;

...

    error = do_get3(cmd, &oh->opheader, &sz);

...

}

...

int
do_get3(int optname, ip_fw3_opheader *op3, size_t *optlen)
{
    int error;

    if (co.test_only)
        return (0);

    if (ipfw_socket == -1)
        ipfw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    if (ipfw_socket < 0)
        err(EX_UNAVAILABLE, "socket");

    op3->opcode = optname;

    error = getsockopt(ipfw_socket, IPPROTO_IP, IP_FW3, op3,
        (socklen_t *)optlen);

    return (error);
}

```

An ipfw3_opheader structure needs to be passed, here is its raw definition ...

```
...

typedef struct ipfw3_opheader {

uint16_t opcode;      (Operation identifier)

        uint16_t version;

... padding ...

};
```

Ipfw3 command list sample (sys/netinet/ip_fw.h):

```
#define          IP_FW_TABLE_XADD      86      /* add entry */

#define           IP_FW_TABLE_XDEL        87      /* delete entry */

#define           IP_FW_TABLE_XGETSIZE    88      /* get table size
(deprecated) */

#define           IP_FW_TABLE_XLIST       89      /* list table contents */

#define           IP_FW_TABLE_XDESTROY    90      /* destroy table */

#define           IP_FW_TABLES_XLIST      92      /* list all tables */

...

#define           IP_FW_DUMP_SOPTCODES    116      /* Dump available
sopts/versions */
```

And here is the list of available opcodes (aka ipfw rules representation):

```
enum ipfw_opcodes {

O_NOP,
```

```

O_IP_SRC,                /* u32 = IP
*/

O_IP_SRC_MASK,           /* ip = IP/mask                */

O_IP_SRC_ME,             /* none
*/

O_IP_SRC_SET,            /* u32=base, arg1=len, bitmap */

O_IP_DST,                /* u32 = IP
*/

O_IP_DST_MASK,           /* ip = IP/mask                */

O_IP_DST_ME,             /* none
*/

O_IP_DST_SET,            /* u32=base, arg1=len, bitmap */

O_IP_SRCPORT,            /* (n)port list:mask 4 byte ea */

O_IP_DSTPORT,            /* (n)port list:mask 4 byte ea */

O_PROTO,                 /* arg1=protocol                */

O_MACADDR2,              /* 2 mac addr:mask              */

O_MAC_TYPE,              /* same as srcport              */

...

};

```

3/ IPFW command from userland to the kernel

Now, let's study, programmatically speaking, an ipfw command and its way to the kernel.

```

/* EXPLANATIONS: Here are the table related command, we will be focusing
on printing them */

```

```

> ipfw table all list

```

```

...

```

```

if (co.use_set || try_next) {

```

```

if (_strcmp(*av, "delete") == 0)

ipfw_delete(av);

else if (_strcmp(*av, "flush") == 0)

ipfw_flush(co.do_force);

else if (_strcmp(*av, "zero") == 0)

ipfw_zero(ac, av, 0 /* IP_FW_ZERO */);

else if (_strcmp(*av, "resetlog") == 0)

ipfw_zero(ac, av, 1 /* IP_FW_RESETLOG */);

/* Here print tables and its alias */

        else if (_strcmp(*av, "print") == 0 ||

                        _strcmp(*av, "list") == 0)

                ipfw_list(ac, av, do_acct);

else if (_strcmp(*av, "show") == 0)

ipfw_list(ac, av, 1 /* show counters */);

else if (_strcmp(*av, "table") == 0)

ipfw_table_handler(ac, av);

else if (_strcmp(*av, "internal") == 0)

ipfw_internal_handler(ac, av);

else

errx(EX_USAGE, "bad command `%s'", *av);

}

...

```

```

void

```

```

ipfw_list(int ac, char *av[], int show_counters)

```

```

{
    ipfw_cfg_lheader *cfg;

    struct format_opts sfo;

    size_t sz;

    ...

    /* get configuration from kernel */

    cfg = NULL;

    sfo.show_counters = show_counters;

    sfo.show_time = co.do_time;

    sfo.flags = IPFW_CFG_GET_STATIC;

    if (co.do_dynamic != 0)

        sfo.flags |= IPFW_CFG_GET_STATES;

    if ((sfo.show_counters | sfo.show_time) != 0)

        sfo.flags |= IPFW_CFG_GET_COUNTERS;

        if (ipfw_get_config(&co, &sfo, &cfg, &sz) != 0)

            err(EX_OSERR, "retrieving config failed");

    ...

}

...

static int

ipfw_get_config(struct cmdline_opts *co, struct format_opts *fo,
ipfw_cfg_lheader **pcfg, size_t *psize)

{
    ipfw_cfg_lheader *cfg;

    size_t sz;

    int i;

```

```

if (co->test_only != 0) {

fprintf(stderr, "Testing only, list disabled\n");

return (0);

}


/* Start with some data size */

sz = 4096;

cfg = NULL;


for (i = 0; i < 16; i++) {

if (cfg != NULL)

free(cfg);

if ((cfg = calloc(1, sz)) == NULL)

return (ENOMEM);


cfg->flags = fo->flags;

cfg->start_rule = fo->first;

cfg->end_rule = fo->last;


/* This is where the command is going to be send to the kernel with a raw
memory estimate and try again by growing it if the data requires more room*/


        if (do_get3(IP_FW_XGET, &cfg->opheader, &sz) != 0) {

if (errno != ENOMEM) {

free(cfg);

return (errno);

}

}

```

```

/* Buffer size is not enough. Try to increase */

sz = sz * 2;

if (sz < cfg->size)

sz = cfg->size;

continue;

}

*pcfg = cfg;

*psize = sz;

return (0);

}

free(cfg);

return (ENOMEM);

}

```

Here, the userland part ends and we're going to see what happens in the kernel:

```

static int

dump_config(struct ip_fw_chain *chain, ip_fw3_opheader *op3,

struct sockopt_data *sd)

{

ipfw_cfg_lheader *hdr;

struct ip_fw *rule;

size_t sz, rnum;

uint32_t hdr_flags;

```

```

int error, i;

struct dump_args da;

uint32_t *bmask;

/* Our data lies contigously in raw form into the ipfw_cfg_header struct

    Below, we ll get the data we re interested in calculating the needed
    memory depending of the flags we passed earlier

*/

hdr = (ipfw_cfg_lheader *)ipfw_get_sopt_header(sd, sizeof(*hdr));

    // Depending on the flags you passed from the command line, various
    data are going to be displayed

if (hdr->flags & IPFW_CFG_GET_STATIC) {

for (i = da.b; i < da.e; i++) {

rule = chain->map[i];

da.rsize += RULEUSIZE1(rule) + sizeof(ipfw_obj_tlv);

da.rcount++;

da.tcount += ipfw_mark_table_kidx(chain, rule, bmask);

}

/* Add counters if requested */

if (hdr->flags & IPFW_CFG_GET_COUNTERS) {

da.rsize += sizeof(struct ip_fw_bcounter) * da.rcount;

da.rcounters = 1;

}

if (da.tcount > 0)

sz += da.tcount * sizeof(ipfw_obj_ntlv) +

sizeof(ipfw_obj_ctlv);

```

```

sz += da.rsize + sizeof(ipfw_obj_ctlv);

}

...

if (hdr->flags & IPFW_CFG_GET_STATES)

sz += ipfw_dyn_get_count() * sizeof(ipfw_obj_dyntlv) +
sizeof(ipfw_obj_ctlv);

...

static int
dump_static_rules(struct ip_fw_chain *chain, struct dump_args *da,
uint32_t *bmask, struct sockopt_data *sd)
{
int error;

int i, l;

uint32_t tcount;

ipfw_obj_ctlv *ctlv;

struct ip_fw *krule;

caddr_t dst;

...

i = 0;

tcount = da->tcount;

while (tcount > 0) {

if ((bmask[i / 32] & (1 << (i % 32))) == 0) {

i++;

continue;

}

```

```

        if ((error = ipfw_objhash_ntlv(chain, i, sd)) != 0)

            return (error);

i++;

tcount--;

}

int
ipfw_export_table_ntlv(struct ip_fw_chain *ch, uint16_t kidx,
struct sockopt_data *sd)
{
    struct namedobj_instance *ni;
    struct named_object *no;
    ipfw_obj_ntlv *ntlv;

    ni = CHAIN_TO_NI(ch);
    no = ipfw_objhash_lookup_kidx(ni, kidx);
    KASSERT(no != NULL, ("invalid table kidx passed"));

    ntlv = (ipfw_obj_ntlv *)ipfw_get_sopt_space(sd, sizeof(*ntlv));
    if (ntlv == NULL)
        return (ENOMEM);

    ntlv->head.type = IPFW_TLV_TBL_NAME;
    ntlv->head.length = sizeof(*ntlv);
    ntlv->idx = no->kidx;
    strncpy(ntlv->name, no->name, sizeof(ntlv->name));

    return (0);
}

```

Now we have a better idea of how structured ipfw works. Basically, sysctl for boolean config values and via the socket for the firewall rules, tables settings and so on. Normally, some ideas might start to emerge for the next module.

In the next and last module, we will have an overview of how it works on the kernel side with the firewall rules and configuration to see how to develop a new type of rules.

Exercises

- To have an additional sysctl configuration point entry, explain which part of the kernel needs to be updated and how (what are the requirements?).
- We saw the list of available opcodes to configure or to get information from the kernel. If it is possible to add one, what are the requirements for the enum `ipfw_opcodes` values? What are the requirements for struct `ipfw_insn` (and derived) structs in term of alignment?
- Considering a new feature to add to ipfw and in case a third party code is used, how should the work be shared between the userland and kernel side?

Module 3: Through The Userland to Kernel Codes

In this module, we'll have an overview of ipfw2 - both userland and kernel side -, and how they interact.

First of all, we will see how to use sysctl we saw in previous modules to set simple values. How to communicate settings to the kernel via a socket; all of it going through the userland to kernel codes.

1/ IPFW command line settings via sysctl

We can find it under the FreeBSD's source code we got from svn in the first module.

```
<source code root path>/sbin/ipfw
```

In the previous module, we saw that it was possible to enable the userland to interact with the kernel via a character-device. IPFW works differently. It enables / disables features via sysctl. If you have done some FreeBSD's programming and are already familiar with syscalls like sysctl, sysctlbyname, and sysctlnametomib, you can jump directly to the next chapter.

IPFW uses sysctlbyname and sysctl. Their signatures are:

```
int sysctlbyname(const char *name, void *oldp, size_t *oldlenp, const void
*newp, size_t newlen);

int sysctl(const int *name, u_int namelen, void *oldp, size_t oldlenp, const
void *newp, size_t newlen);
```

If you wish to get a value, the oldp and oldlenp arguments need to be used. To set a value, use the newp and newlen arguments.

For example, to get the number of CPUs available:

```
int nbcpu;

size_t nbcpulen = sizeof(nbcpu);

...

if (sysctlbyname("hw.ncpu", &nbcpu, &nbcpulen, NULL, 0) == 0)

    printf("%d cpus\n", nbcpu);
```

Alternatively:

```
int mib[2];

...

mib[0] = CTL_HW;

mib[1] = HW_NCPU;

...

if (sysctl(mib, sizeof(mib), &nbcpu, &nbcpulen, NULL, 0) == 0)

    ...
```

To set a value, like the number of maximum file descriptors:

```
int maxfiles = 4096;

size_t maxfileslen = sizeof(maxfiles);

...

if (sysctlbyname("kern.maxfiles", NULL, 0, &maxfiles, maxfileslen) == 0)...

...

mib[0] = CTL_KERN;

mib[1] = KERN_MAXFILES;
```

...

```
if (sysctl(mib, sizeof(mib), NULL, 0, &maxfiles, maxfileslen) == 0)...
```

IPFW uses the sysctl* family of functions to turn the firewall on/off and to make ipfw more verbose.

Here is the part of the code part where the firewall is enabled or disabled in sbin/ipfw/ipfw2.c. This is called by the code in the main() function that parses user-supplied arguments.

```
} else if (_strcmp(*av, "firewall") == 0) {  
  
    sysctlbyname("net.inet.ip.fw.enable", NULL, 0,  
                &which, sizeof(which));  
  
    sysctlbyname("net.inet6.ip6.fw.enable", NULL, 0,  
                &which, sizeof(which));
```

For more information, especially about all possible requests, check the sysctl man page:

```
man 3 sysctl
```

2/ IPFW command line settings via socket

Here, we need to communicate our settings to the kernel side.

ipfw uses a socket to add a rule via an identified optname/command.

Here is a sample of code responsible for getting settings from the kernel from sbin/ipfw/tables.c:

...

```
static int  
table_do_modify_record(int cmd, ipfw_obj_header *oh,  
    ipfw_obj_tentry *tent, int count, int atomic)  
{  
  
    ipfw_obj_ctlv *ctlv;
```

```

    ipfw_obj_tentry *tent_base;

    caddr_t pbuf;

    char xbuf[sizeof(*oh) + sizeof(ipfw_obj_ctlv) + sizeof(*tent)];

    int error, i;

    size_t sz;

...

    error = do_get3(cmd, &oh->opheader, &sz);

...

}

...

int
do_get3(int optname, ip_fw3_opheader *op3, size_t *optlen)
{
    int error;

    if (co.test_only)
        return (0);

    if (ipfw_socket == -1)

/* even though we could have used AF_LOCAL here, we need to distinguish IPV4
from IPV6 matters */

        ipfw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

/* communication with ipfw2 command line here (via do_cmd), the "get config"
command will be coming from there */

    if (ipfw_socket < 0)

        err(EX_UNAVAILABLE, "socket");

```

```

/* here, we send the equivalent programmatically speaking, of the command ;
opcode which is the hexadecimal representation of */

    op3->opcode = optname;

    error = getsockopt(ipfw_socket, IPPROTO_IP, IP_FW3, op3,
        (socklen_t *)optlen);

    return (error);
}

```

An ipfwq3_opheader structure needs to be passed. Here is its raw definition:

```

...

typedef struct ipfw3_opheader {

    uint16_t opcode; (Operation identifier)

    uint16_t version;

    ... padding ...

};

```

Some representative Ipfw3 commands are shown below. These are from sys/netinet/ip_fw.h.

```

#define IP_FW_TABLE_XADD 86 /* add entry */

#define IP_FW_TABLE_XDEL 87 /* delete entry */

#define IP_FW_TABLE_XGETSIZE 88 /* get table size (deprecated) */

#define IP_FW_TABLE_XLIST 89 /* list table contents */

#define IP_FW_TABLE_XDESTROY 90 /* destroy table */

#define IP_FW_TABLES_XLIST 92 /* list all tables */

```

...

```
#define IP_FW_DUMP_SOPTCODES 116 /* Dump available sopts/versions */
```

And here is the list of available opcodes, also from sys/netinet/ip_fw.h:

```
enum ipfw_opcodes {  
    O_NOP,  
  
    O_IP_SRC,          /* u32 = IP          */  
    O_IP_SRC_MASK,     /* ip = IP/mask      */  
    O_IP_SRC_ME,       /* none              */  
    O_IP_SRC_SET,      /* u32=base, arg1=len, bitmap */  
  
    O_IP_DST,          /* u32 = IP          */  
    O_IP_DST_MASK,     /* ip = IP/mask      */  
    O_IP_DST_ME,       /* none              */  
    O_IP_DST_SET,      /* u32=base, arg1=len, bitmap */  
  
    O_IP_SRCPORT,      /* (n)port list:mask 4 byte ea */  
    O_IP_DSTPORT,      /* (n)port list:mask 4 byte ea */  
    O_PROTO,           /* arg1=protocol      */  
  
    O_MACADDR2,        /* 2 mac addr:mask    */  
    O_MAC_TYPE,        /* same as srcport     */  
    ...  
};
```

3/ IPFW command from userland to the kernel

Now, let's study how an ipfw command makes its way to the kernel from sbin/ipfw/ipfw2.c:

```
> ipfw table all list
```

```
...
```

```
    if (co.use_set || try_next) {  
        if (_strcmp(*av, "delete") == 0)  
            ipfw_delete(av);  
        else if (_strcmp(*av, "flush") == 0)  
            ipfw_flush(co.do_force);  
        else if (_strcmp(*av, "zero") == 0)  
            ipfw_zero(ac, av, 0 /* IP_FW_ZERO */);  
        else if (_strcmp(*av, "resetlog") == 0)  
            ipfw_zero(ac, av, 1 /* IP_FW_RESETLOG */);
```

```
/* Here is an example we can go through the code flow from the table  
print/list command */
```

```
    else if (_strcmp(*av, "print") == 0 ||  
            _strcmp(*av, "list") == 0)  
        ipfw_list(ac, av, do_acct);  
    else if (_strcmp(*av, "show") == 0)  
        ipfw_list(ac, av, 1 /* show counters */);  
    else if (_strcmp(*av, "table") == 0)  
        ipfw_table_handler(ac, av);  
    else if (_strcmp(*av, "internal") == 0)  
        ipfw_internal_handler(ac, av);  
    else  
        errx(EX_USAGE, "bad command `%s'", *av);
```

```

    }

...

void
ipfw_list(int ac, char *av[], int show_counters)
{
    ipfw_cfg_lheader *cfg;

    struct format_opts sfo;

    size_t sz;

...

    /* get configuration from kernel */

    cfg = NULL;

    sfo.show_counters = show_counters;

    sfo.show_time = co.do_time;

    sfo.flags = IPFW_CFG_GET_STATIC;

    if (co.do_dynamic != 0)

        sfo.flags |= IPFW_CFG_GET_STATES;

    if ((sfo.show_counters | sfo.show_time) != 0)

        sfo.flags |= IPFW_CFG_GET_COUNTERS;

    /* We get the general config from here */

    if (ipfw_get_config(&co, &sfo, &cfg, &sz) != 0)

        err(EX_OSERR, "retrieving config failed");

...

}

...

static int

ipfw_get_config(struct cmdline_opts *co, struct format_opts *fo,

```

```

ipfw_cfg_lheader **pcfg, size_t *psize)
{
    ipfw_cfg_lheader *cfg;

    size_t sz;

    int i;


    if (co->test_only != 0) {
        fprintf(stderr, "Testing only, list disabled\n");
        return (0);
    }


    /* Start with some data size */

    sz = 4096;

    cfg = NULL;


    for (i = 0; i < 16; i++) {
        if (cfg != NULL)
            free(cfg);

        if ((cfg = calloc(1, sz)) == NULL)
            return (ENOMEM);


        cfg->flags = fo->flags;

        cfg->start_rule = fo->first;

        cfg->end_rule = fo->last;


        if (do_get3(IP_FW_XGET, &cfg->opheader, &sz) != 0) {

```

```

        if (errno != ENOMEM) {

            free(cfg);

            return (errno);

        }

        /* Buffer size is not enough. Try to increase */

        sz = sz * 2;

        if (sz < cfg->size)

            sz = cfg->size;

        continue;

    }

    *pcfg = cfg;

    *psize = sz;

    return (0);

}

free(cfg);

return (ENOMEM);

}

```

Here, the userland part ends and we're going to see what happens in the kernel:

```

static int

dump_config(struct ip_fw_chain *chain, ip_fw3_opheader *op3,

            struct sockopt_data *sd)

{

```

```

ipfw_cfg_lheader *hdr;

struct ip_fw *rule;

size_t sz, rnum;

uint32_t hdr_flags;

int error, i;

struct dump_args da;

uint32_t *bmask;


hdr = (ipfw_cfg_lheader *)ipfw_get_sopt_header(sd, sizeof(*hdr));

// Depending on the flags you passed from the command line, various data
are going to be displayed


if (hdr->flags & IPFW_CFG_GET_STATIC) {

    for (i = da.b; i < da.e; i++) {

        rule = chain->map[i];

        da.rsize += RULEUSIZE1(rule) + sizeof(ipfw_obj_tlv);

        da.rcount++;

        da.tcount += ipfw_mark_table_kidx(chain, rule, bmask);

    }

    /* Add counters if requested */

    if (hdr->flags & IPFW_CFG_GET_COUNTERS) {

        da.rsize += sizeof(struct ip_fw_bcounter) * da.rcount;

        da.rcounters = 1;

    }

    if (da.tcount > 0)

        sz += da.tcount * sizeof(ipfw_obj_ntlv) +

```

```

        sizeof(ipfw_obj_ctlv);

    sz += da.rsize + sizeof(ipfw_obj_ctlv);

}

...

if (hdr->flags & IPFW_CFG_GET_STATES)

    sz += ipfw_dyn_get_count() * sizeof(ipfw_obj_dyntlv) +
        sizeof(ipfw_obj_ctlv);

...

static int
dump_static_rules(struct ip_fw_chain *chain, struct dump_args *da,
    uint32_t *bmask, struct sockopt_data *sd)
{
    int error;

    int i, l;

    uint32_t tcount;

    ipfw_obj_ctlv *ctlv;

    struct ip_fw *krule;

    caddr_t dst;

    ...

    i = 0;

    tcount = da->tcount;

    while (tcount > 0) {

        if ((bmask[i / 32] & (1 << (i % 32))) == 0) {

            i++;

            continue;

```

```

    }

    if ((error = ipfw_export_table_ntlv(chain, i, sd)) != 0)

        return (error);

    i++;

    tcount--;

}

```

```

int

```

```

ipfw_export_table_ntlv(struct ip_fw_chain *ch, uint16_t kidx,
    struct sockopt_data *sd)
{
    struct namedobj_instance *ni;
    struct named_object *no;
    ipfw_obj_ntlv *ntlv;

    ni = CHAIN_TO_NI(ch);

    no = ipfw_objhash_lookup_kidx(ni, kidx);
    KASSERT(no != NULL, ("invalid table kidx passed"));

    ntlv = (ipfw_obj_ntlv *)ipfw_get_sopt_space(sd, sizeof(*ntlv));
    if (ntlv == NULL)

        return (ENOMEM);

    ntlv->head.type = IPFW_TLV_TBL_NAME;

```

```
ntlv->head.length = sizeof(*ntlv);

ntlv->idx = no->kidx;

strncpy(ntlv->name, no->name, sizeof(ntlv->name));

return (0);

}
```

Now, we have a better idea of how ipfw works. Basically, sysctls are used for boolean config values and sockets for firewall rules, tables settings and so on. Normally, some idea might start to emerge in the next module.

In the next and final module, we will have an overview of how it works in the kernel side, the firewall rules and configuration to see how to develop a new type of rules.

Exercises

- To have an additional sysctl configuration point entry, explain which part of the kernel needs to be updated and how (what are the requirements)?
- We saw the list of available opcodes to configure or to get information from the kernel. If it is possible to add one, what are the requirements for the enum `ipfw_opcodes` values? What are the requirements for struct `ipfw_insn` (and derived) structs in terms of alignment?
- Considering a new feature to add to ipfw and in case a third party code is used, how is the work ought to be shared between the userland and kernel side?

Module 4:

DUMMYNET Module

Workflow Study

In this last module, we'll not only look at ipfw's communication with the kernel but also how the firewall configuration and rules are handled.

We will go through the dummynet module, its workflow and how it operates with the kernel so you would be able to add new opcodes on your own.

1/ DUMMYNET module study

The dummynet (unlike the name suggests, it is not a kind of fake/no-op module, but the name is due to historical reasons as it was a test ensemble in the beginning) module allows setting network bandwidth limits (called traffic shaping), and is an optional submodule of ipfw. Since it is optional, it has to be enabled via the kernel configuration options DUMMYNET (into sys/conf/options). Beware there is the compat(ibility) layer for 32 bits (if needed), cloudabi (the secure posix interface layer) and probably for the Linux API compatibility layer that you might need to take care of when you develop a kernel module.

Programmatically, there are four flags available to add a pipe (a pipe is to viewed as a workflow queue where every packet belonging to this same queue will be treated by the scheduler. (For more details into netpfil/ipfw/dummysnext.txt), deleting a pipe, flushing and getting the pipe info.

...

```
#define IP_DUMMYNET_CONFIGURE 60
#define IP_DUMMYNET_DEL      61
#define IP_DUMMYNET_FLUSH    62
```

```
#define IP_DUMMYNET_GET 64
```

...

=> The **ipfw** module ought to be loaded after **ipfw**. The **DN_MODEV_ORD** in **sys/netpfil/ipfw/ip_dummynet.c** ensures this:

```
#define DN_SI_SUB    SI_SUB_PROTO_IFATTACHDOMAIN

/* dummynet is guaranteed to start after ipfw, given that ipfw init phase
occurs at (SI_ORDER_ANY - 255) giving enough room for modules as you can see
*/

#define DN_MODEV_ORD (SI_ORDER_ANY - 128) /* after ipfw */

DECLARE_MODULE(dummynet, dummynet_mod, DN_SI_SUB, DN_MODEV_ORD);

MODULE_DEPEND(dummynet, ipfw, 3, 3, 3);

MODULE_VERSION(dummynet, 3);
```

Now, let's see the DUMMYNET's configuration part. First, copy the sockopt data from userland to kernel first.

Then make the configuration in FreeBSD 7.x or FreeBSD 8.x format. For backward compatibility, the old FreeBSD 7 syntax is still supported (the new syntax is much less error prone/buggy, more consistent (pipe usage)).

...

```
case IP_DUMMYNET_CONFIGURE:

    v = malloc(len, M_TEMP, M_WAITOK);

    error = sooptcopyin(sopt, v, len, len);

    if (error)

        break;

    error = dn_compat_configure(v);

    free(v, M_TEMP);
```

```
break;
```

```
...
```

```
static int
```

```
dn_compat_configure(void *v)
```

```
{
```

```
    struct dn_id *buf = NULL, *base;
```

```
    struct dn_sch *sch = NULL;
```

```
    struct dn_link *p = NULL;
```

```
    struct dn_fs *fs = NULL;
```

```
    struct dn_profile *pf = NULL;
```

```
    int lmax;
```

```
    int error;
```

=> Those two struct represent FreeBSD 7 and 8 pipe configuration format

```
struct dn_pipe7 *p7 = (struct dn_pipe7 *)v;
```

```
struct dn_pipe8 *p8 = (struct dn_pipe8 *)v;
```

```
...
```

=> If we have a pipe to configure:

```
i = p7->pipe_nr;
```

```
if (i != 0) { /* pipe config */
```

=> We take chunks of the buffer:

```
    sch = o_next(&buf, sizeof(*sch), DN_SCH);
```

```
    p = o_next(&buf, sizeof(*p), DN_LINK);
```

```
    fs = o_next(&buf, sizeof(*fs), DN_FS);
```

```
    error = dn_compat_config_pipe(sch, p, fs, v);
```

...

=> **Here, we carry out traffic shaping configuration, bandwidth, delay and burst.s**

```
static int
dn_compat_config_pipe(struct dn_sch *sch, struct dn_link *p,
                     struct dn_fs *fs, void* v)
{
    struct dn_pipe7 *p7 = (struct dn_pipe7 *)v;
    struct dn_pipe8 *p8 = (struct dn_pipe8 *)v;
    int i = p7->pipe_nr;

    sch->sched_nr = i;
    sch->oid.subtype = 0;
    p->link_nr = i;
    fs->fs_nr = i + 2*DN_MAX_ID;
    fs->sched_nr = i + DN_MAX_ID;

    /* Common to 7 and 8 */
    p->bandwidth = p7->bandwidth;
    p->delay = p7->delay;
    if (!is7) {
        /* FreeBSD 8 has burst */
        p->burst = p8->burst;
    }

    /* fill the fifo flowset */
    dn_compat_config_queue(fs, v);
}
```

```

fs->fs_nr = i + 2*DN_MAX_ID;

fs->sched_nr = i + DN_MAX_ID;


/* Move scheduler related parameter from fs to sch */
sch->buckets = fs->buckets; /*XXX*/

fs->buckets = 0;

if (fs->flags & DN_HAVE_MASK) {

    sch->flags |= DN_HAVE_MASK;

    fs->flags &= ~DN_HAVE_MASK;

    sch->sched_mask = fs->flow_mask;

    bzero(&fs->flow_mask, sizeof(struct ipfw_flow_id));

}

return 0;

}

```

=> **Once the bandwidth is set (and eventually the extra burst allowance), these settings are used here.**

```

if (si->idle_time < dn_cfg.curr_time) {

    /* Do this only on the first packet on an idle pipe */

    struct dn_link *p = &fs->sched->link;

    si->sched_time = dn_cfg.curr_time;

    si->credit = dn_cfg.io_fast ? p->bandwidth : 0;

    if (p->burst) {

        uint64_t burst = (dn_cfg.curr_time - si->idle_time) * p->bandwidth;

```

```

    if (burst > p->burst)

        burst = p->burst;

    si->credit += burst;

}

}

```

=> Here, the delaying / bandwidth limit policies will be applied before giving back the upper hand

to ipfw through `dummysnet_send`.

```

m = serve_sched(NULL, si, dn_cfg.curr_time);

...

static void
dummysnet_send(struct mbuf *m)
{
    struct mbuf *n;

    for (; m != NULL; m = n) {
        struct ifnet *ifp = NULL; /* gcc 3.4.6 complains */
        struct m_tag *tag;

        int dst;

        ...
    }
}

```

=> **IPFW actions**

```

switch (dst) {

case DIR_OUT:

    ip_output(m, NULL, NULL, IP_FORWARDING, NULL, NULL);

    break ;
}

```

```

case DIR_IN :

    netisr_dispatch(NETISR_IP, m);

    break;


#ifdef INET6

case DIR_IN | PROTO_IPV6:

    netisr_dispatch(NETISR_IPV6, m);

    break;


case DIR_OUT | PROTO_IPV6:

    ip6_output(m, NULL, NULL, IPV6_FORWARDING, NULL, NULL,
NULL);

    break;

#endif


case DIR_FWD | PROTO_IFB: /* DN_TO_IFB_FWD: */

    if (bridge_dn_p != NULL)

        ((*bridge_dn_p)(m, ifp));

    else

        printf("dummynet: if_bridge not loaded\n");

    break;


case DIR_IN | PROTO_LAYER2: /* DN_TO_ETH_DEMUX: */

    /*

    * The Ethernet code assumes the Ethernet header is

    * contiguous in the first mbuf header.

```

```

    * Ensure this is true.

    */

    if (m->m_len < ETHER_HDR_LEN &&
        (m = m_pullup(m, ETHER_HDR_LEN)) == NULL) {
        printf("dummynet/ether: pullup failed, "
            "dropping packet\n");
        break;
    }

    ether_demux(m->m_pkthdr.rcvif, m);
    break;

case DIR_OUT | PROTO_LAYER2: /* N_TO_ETH_OUT: */
    ether_output_frame(ifp, m);
    break;

case DIR_DROP:
    /* drop the packet after some time */
    FREE_PKT(m);
    break;

default:
    printf("dummynet: bad switch %d!\n", dst);
    FREE_PKT(m);
    break;
}

...

```

2/ IPFW firewall rule route study

Now, let's see how a firewall rule is processed in the kernel.

First, the rules parser sockopt copies from userland to kernel, and checks the rule's validity.

```
int
ipfw_ctl(struct sockopt *sopt)
{
#define RULE_MAXSIZE (512*sizeof(u_int32_t))

    int error;

    size_t size, valsize;

    struct ip_fw *buf;

    struct ip_fw_rule0 *rule;

    struct ip_fw_chain *chain;

    u_int32_t rulenum[2];

    uint32_t opt;

    struct rule_check_info ci;

    IPFW_RLOCK_TRACKER;

    chain = &V_layer3_chain;

    error = 0;

    /* Save original valsize before it is altered via sooptcopyin() */
    valsize = sopt->sopt_valsize;

    opt = sopt->sopt_name;

    ...
```

```
case IP_FW_ADD:
```

```
    rule = malloc(RULE_MAXSIZE, M_TEMP, M_WAITOK);
```

=> **Here, we must check if it is FreeBSD 7.x rule format ; sopt_valsize field will give this hint after the rule is copied in the kernel. Then, it will be converted to FreeBSD 8.x format.**

```
    error = sooptcopyin(sopt, rule, RULE_MAXSIZE,
        sizeof(struct ip_fw7) );
```

```
    ...
```

```
    if (error == 0)
```

```
        error = check_ipfw_rule0(rule, size, &ci);
```

```
    if (error == 0) {
```

```
        /* locking is done within add_rule() */
```

```
        struct ip_fw *krule;
```

```
        krule = ipfw_alloc_rule(chain, RULEKSIZE0(rule));
```

```
        ci.urule = (caddr_t)rule;
```

```
        ci.krule = krule;
```

```
        import_rule0(&ci);
```

```
        error = commit_rules(chain, &ci, 1);
```

=> **If the userland requested an answer, it is converted back to FreeBSD 7.x format when necessary.**

```
    if (!error && sopt->sopt_dir == SOPT_GET) {
```

```
        if (is7) {
```

```
            error = convert_rule_to_7(rule);
```

```
            size = RULESIZE7(rule);
```

```
            if (error) {
```

```
                free(rule, M_TEMP);
```

```
                return error;
```

```

    }

}

```

=> **Sending back the rule data to the userland:**

```

        error = sooptcopyout(sopt, rule, size);

    }

}

```

Afterwards, IPFW has main firewall rules to check where it is decided if a packet ought to be accepted, dropped, passed to dumynet module, and so on.

```

int

ipfw_chk(struct ip_fw_args *args)

{

    ...

    /* Identify IP packets and fill up variables. */

```

=> **Similar checking is done on ipv4.**

```

if (pktlen >= sizeof(struct ip6_hdr) &&

    (args->eh == NULL || etype == ETHERTYPE_IPV6) && ip->ip_v == 6) {

    struct ip6_hdr *ip6 = (struct ip6_hdr *)ip;

    is_ipv6 = 1;

    args->f_id.addr_type = 6;

    hlen = sizeof(struct ip6_hdr);

    proto = ip6->ip6_nxt;


    /* Search extension headers to find upper layer protocols */

    while (ulp == NULL && offset == 0) {

        switch (proto) {

```

```

case IPPROTO_ICMPV6:

    PULLUP_TO(hlen, ulp, struct icmp6_hdr);

    icmp6_type = ICMP6(ulp)->icmp6_type;

    break;


case IPPROTO_TCP:

    PULLUP_TO(hlen, ulp, struct tcphdr);

    dst_port = TCP(ulp)->th_dport;

    src_port = TCP(ulp)->th_sport;

    /* save flags for dynamic rules */

    args->f_id._flags = TCP(ulp)->th_flags;

    break;


case IPPROTO_SCTP:

    PULLUP_TO(hlen, ulp, struct sctphdr);

    src_port = SCTP(ulp)->src_port;

    dst_port = SCTP(ulp)->dest_port;

    break;


case IPPROTO_UDP:

    PULLUP_TO(hlen, ulp, struct udphdr);

    dst_port = UDP(ulp)->uh_dport;

    src_port = UDP(ulp)->uh_sport;

    break;


...

```

=> Here comes an important part of your future custom module building, the check of the known rules where each packet is inspected from the following loop:

```
for (; f_pos < chain->n_rules; f_pos++) {

    ipfw_insn *cmd;

    uint32_t tablearg = 0;

    int l, cmdlen, skip_or; /* skip rest of OR block */

    struct ip_fw *f;

    f = chain->map[f_pos];

    if (V_set_disable & (1 << f->set) )

        continue;

    skip_or = 0;

    for (l = f->cmd_len, cmd = f->cmd ; l > 0 ;

        l -= cmdlen, cmd += cmdlen) {

        int match;

        /*

         * check_body is a jump target used when we find a

         * CHECK_STATE, and need to jump to the body of

         * the target rule.

         */

        /* check_body: */

        cmdlen = F_LEN(cmd);

        /*
```

```

* An OR block (insn_1 || .. || insn_n) has the
* F_OR bit set in all but the last instruction.
* The first match will set "skip_or", and cause
* the following instructions to be skipped until
* past the one with the F_OR bit clear.
*/

if (skip_or) {          /* skip this instruction */
    if ((cmd->len & F_OR) == 0)
        skip_or = 0; /* next one is good */
    continue;
}

match = 0; /* set to 1 if we succeed */

switch (cmd->opcode) {
/*
* The first set of opcodes compares the packet's
* fields with some pattern, setting 'match' if a
* match is found. At the end of the loop, there is
* logic to deal with F_NOT and F_OR flags associated
* with the opcode.
*/

case O_NOP:
    match = 1;
    break;

case O_FORWARD_MAC:
    printf("ipfw: opcode %d unimplemented\n",

```

```

        cmd->opcode);

    break;

case O_GID:

case O_UID:

case O_JAIL:

    /*
     * We only check offset == 0 && proto != 0,
     * as this ensures that we have a
     * packet with the ports info.
     */

    if (offset != 0)

        break;

    if (proto == IPPROTO_TCP ||

        proto == IPPROTO_UDP)

        match = check_uidgid(

            (ipfw_insn_u32 *)cmd,

            args, &ucred_lookup,

#ifdef __FreeBSD__

            &ucred_cache);

#else

            (void *)&ucred_cache);

#endif

        break;

case O_RECV:

    match = iface_match(m->m_pkthdr.rcvif,

```

```

        (ipfw_insn_if *)cmd, chain, &tablearg);

break;

...

case O_MAC_TYPE:

    if (args->eh != NULL) {

        u_int16_t *p =

            ((ipfw_insn_u16 *)cmd)->ports;

        int i;

        for (i = cmdlen - 1; !match && i>0;

            i--, p += 2)

            match = (etype >= p[0] &&

                etype <= p[1]);

    }

    break;

case O_FRAG:

    match = (offset != 0);

    break;

case O_IN:    /* "out" is "not in" */

    match = (oif == NULL);

    break;

case O_LAYER2:

    match = (args->eh != NULL);

```

```

        break;

case O_DIVERTED:
    {
        /* For diverted packets, args->rule.info
         * contains the divert port (in host format)
         * reason and direction.
         */

        uint32_t i = args->rule.info;

        match = (i & IPFW_IS_MASK) == IPFW_IS_DIVERT &&
            cmd->arg1 & ((i & IPFW_INFO_IN) ? 1 : 2);
    }

    break;

case O_PROTO:
    /*
     * We do not allow an arg of 0, so the
     * check of "proto" only suffices.
     */

    match = (proto == cmd->arg1);

    break;

case O_IP_SRC:
    match = is_ipv4 &&
        (((ipfw_insn_ip *)cmd)->addr.s_addr ==
        src_ip.s_addr);

    break;

```

```

case O_IP_SRC_LOOKUP:

case O_IP_DST_LOOKUP:

    if (is_ipv4) {

        uint32_t key =

            (cmd->opcode == O_IP_DST_LOOKUP) ?

                dst_ip.s_addr : src_ip.s_addr;

        uint32_t v = 0;


        if (cmdlen > F_INSN_SIZE(ipfw_insn_u32)) {

            /* generic lookup. The key must be

             * in 32bit big-endian format.

             */

            v = ((ipfw_insn_u32 *)cmd)->d[1];

            if (v == 0)

                key = dst_ip.s_addr;

            else if (v == 1)

                key = src_ip.s_addr;

            else if (v == 6) /* dscp */

                key = (ip->ip_tos >> 2) & 0x3f;

            else if (offset != 0)

                break;

            else if (proto != IPPROTO_TCP &&

                proto != IPPROTO_UDP)

                break;

            else if (v == 2)

                key = dst_port;

```

```

else if (v == 3)

    key = src_port;

#ifdef USERSPACE

else if (v == 4 || v == 5) {

    check_uidgid(

        (ipfw_insn_u32 *)cmd,

        args, &ucred_lookup,

#ifdef __FreeBSD__

        &ucred_cache);

        if (v == 4 /* O_UID */)

            key = ucred_cache->cr_uid;

        else if (v == 5 /* O_JAIL */)

            key = ucred_cache->cr_prison->pr_id;

#else /* !__FreeBSD__ */

        (void *)&ucred_cache);

        if (v == 4 /* O_UID */)

            key = ucred_cache.uid;

        else if (v == 5 /* O_JAIL */)

            key = ucred_cache.xid;

#endif /* !__FreeBSD__ */

    } else

#endif /* !USERSPACE */

        break;

    }

    match = ipfw_lookup_table(chain,

        cmd->arg1, key, &v);

    if (!match)

```

```

        break;

    if (cmdlen == F_INSN_SIZE(ipfw_insn_u32))

        match =

            ((ipfw_insn_u32 *)cmd)->d[0] == v;

    else

        tablearg = v;

} else if (is_ipv6) {

    uint32_t v = 0;

    void *pkey = (cmd->opcode == O_IP_DST_LOOKUP) ?

        &args->f_id.dst_ip6: &args->f_id.src_ip6;

    match = ipfw_lookup_table_extended(chain,

        cmd->arg1,

        sizeof(struct in6_addr),

        pkey, &v);

    if (cmdlen == F_INSN_SIZE(ipfw_insn_u32))

        match = ((ipfw_insn_u32 *)cmd)->d[0] == v;

    if (match)

        tablearg = v;

}

break;

...

case O_IP_SRC_MASK:

case O_IP_DST_MASK:

    if (is_ipv4) {

        uint32_t a =

            (cmd->opcode == O_IP_DST_MASK) ?

```

```

        dst_ip.s_addr : src_ip.s_addr;

uint32_t *p = ((ipfw_insn_u32 *)cmd)->d;

int i = cmdlen-1;


    for (; !match && i>0; i-= 2, p+= 2)

        match = (p[0] == (a & p[1]));

    }

break;


case O_IP_SRC_ME:

    if (is_ipv4) {

        struct ifnet *tif;


        INADDR_TO_IFP(src_ip, tif);

        match = (tif != NULL);

        break;

    }


#ifdef INET6

    /* FALLTHROUGH */

case O_IP6_SRC_ME:

    match= is_ipv6 && search_ip6_addr_net(&args->f_id.src_ip6);

#endif

break;


case O_IP_DST_SET:

case O_IP_SRC_SET:

    if (is_ipv4) {

```

```

u_int32_t *d = (u_int32_t *) (cmd+1);

u_int32_t addr =

    cmd->opcode == O_IP_DST_SET ?

    args->f_id.dst_ip :

    args->f_id.src_ip;

    if (addr < d[0])

        break;

    addr -= d[0]; /* subtract base */

    match = (addr < cmd->arg1) &&

        ( d[ 1 + (addr>>5)] &

            (1<<(addr & 0x1f)) );

}

break;

case O_IP_DST:

    match = is_ipv4 &&

        (((ipfw_insn_ip *) cmd)->addr.s_addr ==

            dst_ip.s_addr);

    break;

case O_IP_DST_ME:

    if (is_ipv4) {

        struct ifnet *tif;

        INADDR_TO_IFP(dst_ip, tif);

        match = (tif != NULL);

```

```

        break;

    }

#ifdef INET6

    /* FALLTHROUGH */

case O_IP6_DST_ME:

    match= is_ipv6 && search_ip6_addr_net(&args->f_id.dst_ip6);

#endif

    break;


case O_IP_SRCPORT:
case O_IP_DSTPORT:

    /*

    * offset == 0 && proto != 0 is enough

    * to guarantee that we have a

    * packet with port info.

    */

    if ((proto==IPPROTO_UDP || proto==IPPROTO_TCP)

        && offset == 0) {

        u_int16_t x =

            (cmd->opcode == O_IP_SRCPORT) ?

            src_port : dst_port ;

        u_int16_t *p =

            ((ipfw_insn_u16 *)cmd)->ports;

        int i;


        for (i = cmdlen - 1; !match && i>0;

```

```

        i--, p += 2)

        match = (x>=p[0] && x<=p[1]);

    }

    break;

case O_ICMPTYPE:

    match = (offset == 0 && proto==IPPROTO_ICMP &&

        icmp_type_match(ICMP(ulp), (ipfw_insn_u32 *)cmd) );

    break;

...

case O_LOG:

    ipfw_log(chain, f, hlen, args, m,

        oif, offset | ip6f_mf, tablearg, ip);

    match = 1;

    break;

...

case O_ANTISPOOF:

    /* Outgoing packets automatically pass/match */

    if (oif == NULL && hlen > 0 &&

        ( (is_ipv4 && in_localaddr(src_ip))

#ifdef INET6

        || (is_ipv6 &&

            in6_localaddr(&(args->f_id.src_ip6)))

#endif

    )

    )

```

```

    ))

    match =

#ifdef INET6

        is_ipv6 ? verify_path6(

            &(args->f_id.src_ip6),

            m->m_pkthdr.rcvif,

            args->f_id.fib) :

#endif

        verify_path(src_ip,

            m->m_pkthdr.rcvif,

            args->f_id.fib);

    else

        match = 1;

        break;

case O_IPSEC:

#ifdef IPSEC

    match = (m_tag_find(m,

        PACKET_TAG_IPSEC_IN_DONE, NULL) != NULL);

#endif

    /* otherwise no match */

    break;

#ifdef INET6

case O_IP6_SRC:

    match = is_ipv6 &&

        IN6_ARE_ADDR_EQUAL(&args->f_id.src_ip6,

```

```

        &((ipfw_insn_ip6 *)cmd)->addr6);

break;

case O_IP6_DST:

    match = is_ipv6 &&

    IN6_ARE_ADDR_EQUAL(&args->f_id.dst_ip6,

        &((ipfw_insn_ip6 *)cmd)->addr6);

    break;

case O_IP6_SRC_MASK:

case O_IP6_DST_MASK:

    if (is_ipv6) {

        int i = cmdlen - 1;

        struct in6_addr p;

        struct in6_addr *d =

            &((ipfw_insn_ip6 *)cmd)->addr6;

        for (; !match && i > 0; d += 2,

            i -= F_INSN_SIZE(struct in6_addr)

            * 2) {

            p = (cmd->opcode ==

                O_IP6_SRC_MASK) ?

                args->f_id.src_ip6:

                args->f_id.dst_ip6;

            APPLY_MASK(&p, &d[1]);

            match =

                IN6_ARE_ADDR_EQUAL(&d[0],

                    &p);

```

```

        }

    }

    break;

case O_FLOW6ID:

    match = is_ipv6 &&

        flow6id_match(args->f_id.flow_id6,

            (ipfw_insn_u32 *) cmd);

    break;

case O_EXT_HDR:

    match = is_ipv6 &&

        (ext_hd & ((ipfw_insn *) cmd)->arg1);

    break;

case O_IP6:

    match = is_ipv6;

    break;

#endif

```

```

case O_IP4:

    match = is_ipv4;

    break;

case O_TAG: {

    struct m_tag *mtag;

    uint32_t tag = TARG(cmd->arg1, tag);

```

```

/* Packet is already tagged with this tag? */
mtag = m_tag_locate(m, MTAG_IPFW, tag, NULL);

/* We have `untag' action when F_NOT flag is
 * present. And we must remove this mtag from
 * mbuf and reset `match' to zero (`match' will
 * be inversed later).
 * Otherwise, we should allocate new mtag and
 * push it into mbuf.
 */
if (cmd->len & F_NOT) { /* `untag' action */
    if (mtag != NULL)
        m_tag_delete(m, mtag);
    match = 0;
} else {
    if (mtag == NULL) {
        mtag = m_tag_alloc( MTAG_IPFW,
            tag, 0, M_NOWAIT);
        if (mtag != NULL)
            m_tag_prepend(m, mtag);
    }
    match = 1;
}
break;
}

```

...

```
case O_TAGGED: {

    struct m_tag *mtag;

    uint32_t tag = TARG(cmd->arg1, tag);

    if (cmdlen == 1) {

        match = m_tag_locate(m, MTAG_IPFW,

            tag, NULL) != NULL;

        break;

    }

    /* we have ranges */

    for (mtag = m_tag_first(m);

        mtag != NULL && !match;

        mtag = m_tag_next(m, mtag)) {

        uint16_t *p;

        int i;

        if (mtag->m_tag_cookie != MTAG_IPFW)

            continue;

        p = ((ipfw_insn_u16 *)cmd)->ports;

        i = cmdlen - 1;

        for(; !match && i > 0; i--, p += 2)

            match =

                mtag->m_tag_id >= p[0] &&
```

```

        mtag->m_tag_id <= p[1];

    }

    break;

}

...

case O_LIMIT:

...

case O_ACCEPT:

    retval = 0;  /* accept */

    l = 0;       /* exit inner loop */

    done = 1;    /* exit outer loop */

    break;

case O_PIPE:

case O_QUEUE:

    set_match(args, f_pos, chain);

    args->rule.info = TARG(cmd->arg1, pipe);

    if (cmd->opcode == O_PIPE)

        args->rule.info |= IPFW_IS_PIPE;

    if (V_fw_one_pass)

        args->rule.info |= IPFW_ONEPASS;

    retval = IP_FW_DUMMYNET;

    l = 0;       /* exit inner loop */

    done = 1;    /* exit outer loop */

    break;

```

```

case O_DIVERT:

case O_TEE:

    if (args->eh) /* not on layer 2 */

        break;

    /* otherwise, this is terminal */

    l = 0;          /* exit inner loop */

    done = 1;       /* exit outer loop */

    retval = (cmd->opcode == O_DIVERT) ?

        IP_FW_DIVERT : IP_FW_TEE;

    set_match(args, f_pos, chain);

    args->rule.info = TARG(cmd->arg1, divert);

    break;

...

case O_REJECT:

    /*

    * Drop the packet and send a reject notice

    * if the packet is not ICMP (or is an ICMP

    * query), and it is not multicast/broadcast.

    */

    if (hlen > 0 && is_ipv4 && offset == 0 &&

        (proto != IPPROTO_ICMP ||

         is_icmp_query(ICMP(ulp))) &&

        !(m->m_flags & (M_BCAST|M_MCAST)) &&

        !IN_MULTICAST(ntohl(dst_ip.s_addr))) {

        send_reject(args, cmd->arg1, ipplen, ip);

```

```

        m = args->m;

    }

    /* FALLTHROUGH */

...

case O_DENY:

    retval = IP_FW_DENY;

    l = 0;          /* exit inner loop */

    done = 1;       /* exit outer loop */

    break;


case O_FORWARD_IP:

    if (args->eh)     /* not valid on layer2 pkts */

        break;

    if (q == NULL || q->rule != f ||

        dyn_dir == MATCH_FORWARD) {

        struct sockaddr_in *sa;

        sa = &(((ipfw_insn_sa *)cmd)->sa);

        if (sa->sin_addr.s_addr == INADDR_ANY) {

#ifdef INET6

            /*

            * We use O_FORWARD_IP opcode for

            * fwd rule with tablearg, but tables

            * now support IPv6 addresses. And

            * when we are inspecting IPv6 packet,

            * we can use nh6 field from

            * table_value as next_hop6 address.

```

```

    */

if (is_ipv6) {

    struct sockaddr_in6 *sa6;

    sa6 = args->next_hop6 =
        &args->hopstore6;

    sa6->sin6_family = AF_INET6;
    sa6->sin6_len = sizeof(*sa6);
    sa6->sin6_addr = TARG_VAL(
        chain, tablearg, nh6);

    /*
     * Set sin6_scope_id only for
     * link-local unicast addresses.
     */

    if (IN6_IS_ADDR_LINKLOCAL(
        &sa6->sin6_addr))

        sa6->sin6_scope_id =
            TARG_VAL(chain,
                tablearg,
                zoneid);

    } else

#endif

{

    sa = args->next_hop =
        &args->hopstore;

    sa->sin_family = AF_INET;
    sa->sin_len = sizeof(*sa);

```

```

        sa->sin_addr.s_addr = htonl(
            TARG_VAL(chain, tablearg,
                nh4));
    }
} else {
    args->next_hop = sa;
}
}

retval = IP_FW_PASS;

l = 0;          /* exit inner loop */
done = 1;       /* exit outer loop */
break;

...

case O_NAT:

    l = 0;          /* exit inner loop */
    done = 1;       /* exit outer loop */

    if (!IPFW_NAT_LOADED) {
        retval = IP_FW_DENY;
        break;
    }

    struct cfg_nat *t;

    int nat_id;

    set_match(args, f_pos, chain);

```

```

/* Check if this is 'global' nat rule */
if (cmd->arg1 == 0) {
    retval = ipfw_nat_ptr(args, NULL, m);
    break;
}

t = ((ipfw_insn_nat *)cmd)->nat;
if (t == NULL) {
    nat_id = TARG(cmd->arg1, nat);
    t = (*lookup_nat_ptr)(&chain->nat, nat_id);

    if (t == NULL) {
        retval = IP_FW_DENY;
        break;
    }

    if (cmd->arg1 != IP_FW_TARG)
        ((ipfw_insn_nat *)cmd)->nat = t;
}

retval = ipfw_nat_ptr(args, t, m);
break;

...

```

=> **We could add additional new type of rules. as we've seen in the previous modules, new opcode are needed.**

```

default:

    panic("-- unknown opcode %d\n", cmd->opcode);

} /* end of switch() on opcodes */

...

```

Final exercise

Now is the time to use all the acquired knowledge you learned through all the modules.

The goal is to add firewall policy (for example, based on IP / country codes, adding the detection algorithm in the kernel side then the rule config from userland perspective), updating the ipfw command line accordingly and above all, the kernel side (preferably as a distinct ipfw kernel module).

David Carlier
email: devnexen@gmail.com